

A Programming Model and Runtime System for Significance-Aware Energy-Efficient Computing

Vassilis Vassiliadis¹ Konstantinos Parasyris² Charalambos Chalios³ Christos D. Antonopoulos⁴
 Spyros Lalīs⁵ Nikolaos Bellas⁶ Hans Vandierendonck⁷ Dimitrios S. Nikolopoulos⁸
^{1,2,4,5,6}Electrical and Computer Eng. Dept. ^{1,2,4,5,6}Centre for Research and Technology Hellas ^{3,7,8}Queen's University Belfast
 University of Thessaly, Greece (CE.R.T.H.), Greece United Kingdom
 {vasiliad¹,koparasy²,cda⁴,lalis⁵,nbellas⁶}@uth.gr {cchalios01³,h.vandierendonck⁷,d.nikolopoulos⁸}@qub.ac.uk

Abstract

Reducing energy consumption is one of the key challenges in computing technology. One factor that contributes to high energy consumption is that all parts of the program are considered equally significant for the accuracy of the end-result. However, in many cases, parts of computations can be performed in an approximate way, or even dropped, without affecting the quality of the final output to a significant degree.

In this paper, we introduce a task-based programming model and runtime system that exploit this observation to trade off the quality of program outputs for increased energy-efficiency. This is done in a structured and flexible way, allowing for easy exploitation of different execution points in the quality/energy space, without code modifications and without adversely affecting application performance. The programmer specifies the significance of tasks, and optionally provides approximations for them. Moreover, she provides hints to the runtime on the percentage of tasks that should be executed accurately in order to reach the target quality of results. The runtime system can apply a number of different policies to decide whether it will execute each individual less-significant task in its accurate form, or in its approximate version. Policies differ in terms of their runtime overhead but also the degree to which they manage to execute tasks according to the programmer's specification.

The results from experiments performed on top of an Intel-based multicore/multiprocessor platform show that, depending on the runtime policy used, our system can achieve an energy reduction of up to 83% compared with a fully accurate execution and up to 35% compared with an approximate version employing loop perforation. At the same time, our approach always results in graceful quality degradation.

Keywords Energy Saving, Approximate Computing, Controlled Quality Degradation, Programming Model, Runtime System Evaluation.

1. Introduction

Energy consumption has become a major barrier, not only for tetherless computing – the traditional energy-constrained environment – but also for other computing domains, including big science. Building an exascale machine with today's technology is impractical due to the inordinate power draw it would require, hampering large-scale scientific efforts. Likewise, current technologies are too energy-inefficient to realize smaller and smarter embedded/wearable devices for a wide range of ubiquitous computing applications that can greatly benefit society, such as personalized health systems.

One factor that contributes to the energy footprint of current computer technology is that all parts of the program are considered to be equally important, and thus are all executed with full accuracy. However, as shown by previous work on approximate computing, in several classes of computations, not all parts or execution phases of a program affect the quality of its output equivalently. In fact, the output may remain virtually unaffected even if some computations produce incorrect results or fail completely. Data intensive applications and kernels from multimedia, data mining, and visualization algorithms, can all tolerate a certain degree of imprecision in their computations. For example, Discrete Cosine Transform (DCT), a module of popular video compression kernels, which transforms a block of image pixels to a block of frequency coefficients, can be partitioned into layers of significance, owing to the fact that human eye is more sensitive to lower spatial frequencies, rather than higher ones. By explicitly tagging operations that contribute to the computation of higher frequencies as less-significant, one can leverage smart underlying system software to trade-off video quality with energy and performance improvements.

In this paper, we introduce a novel, significance-driven programming environment for approximate computing, comprising a programming model, compilation toolchain and runtime system. The environment allows programmers to trade-off the quality of program outputs for increased energy-efficiency, in a structured and flexible way. The programming model follows a task-based approach. For each task, the developer declares its significance depending on how strongly the task contributes to the quality of the final program output, and provides an approximate version of lower complexity that returns a less accurate result or just a meaningful default value. Also, the developer controls the degradation of output quality, by specifying the percentage of tasks to be executed accurately. In turn, the runtime system executes tasks on available cores in a significance-aware fashion, by employing the approximate versions of less-significant tasks, or dropping such tasks altogether. This can lead to shorter makespans and thus to more energy-

efficient executions, without having a significant impact on the results of the computation.

The main contributions of this paper are the following: (i) We propose a new programming model that allows the developer to structure the computation in terms of distinct tasks with different levels of significance, to supply approximate task versions, and to control the degradation of program outputs; (ii) We introduce different runtime policies for deciding which tasks to execute accurately to meet the programmer’s specification; (iii) We implement compiler and runtime support for the programming model and the runtime policies. (iv) We experimentally evaluate the potential of our approach, as well as the performance of the runtime policies.

Previous work has already explored the potential of approximate computing for specific algorithms and software blocks. Our work is largely complementary to these efforts, as we introduce a programming model that makes it possible to apply such techniques in task-based programs that can exploit the parallelism of modern many-core platforms. There are also major differences with other approximate computing frameworks. For instance, the granularity of approximation is at the level of tasks, rather than individual data types, variables or arithmetic operations. Our programming model operates not only at a different granularity but also at a different level of abstraction for approximate computing –relative significance of code blocks–, which enables the compiler and runtime system to implement different policies that trade energy savings with quality. Also, one can explore different points in the quality/energy space in an easy and direct way, without code modifications, simply by specifying the percentage of tasks that should be executed accurately - this can be an open parameter of a kernel or an entire application, which can take different values in each invocation, or be changed interactively by the user.

The rest of the paper is structured as follows. Section 2 introduces the programming model. Section 3 discusses the runtime system, and the different policies used to drive task execution. Section 4 presents the experimental evaluation on top of an Intel-based 16-way multiprocessor/multicore platform, using a set of benchmark kernels that were ported to our programming model. Section 5 gives an overview of related work. Finally, Section 6 concludes the paper and identifies directions for future work.

2. Programming Model

Improving energy consumption by controllably reducing the quality of application output has been already identified as an attractive option in the domain of power-sensitive HPC programming. Part of the problem of energy inefficiency is that all computations are treated as equally important, despite the fact that only a subset of these computations may be critical in order to achieve an acceptable quality of service (QoS). A key challenge though is how to identify and tag computations of the program which must be executed accurately from those that are of less importance and thus can be executed approximately.

In this section we introduce a programming model that allows the programmer to express her perspective on the significance of the contribution of each computation to the quality of the final output. Highly significant computations are executed accurately, whereas non-significant computations can be executed approximately, at the expense of errors, or can be totally dropped.

Our vision is to elevate significance characterization as a first class concern in software development, similar to parallelism and other algorithmic properties traditionally being in the focus of programmers. To this end, the main objectives of the proposed programming model are the following:

- to allow programmers to express the significance of computations in terms of their contribution to the quality of the end-result;
- to allow programmers to specify approximate alternatives for selected computations;
- to allow programmers to express parallelism, beyond significance;
- to allow programmers to control the balance between energy consumption and the quality of the end-result, without sacrificing performance;
- to enable optimization and easy exploration of trade-offs at execution time;
- to be user friendly and architecture agnostic.

Programmers express significance semantics using *#pragma* compiler directives. Pragmas-based programming models facilitate non-invasive and progressive code transformations, without requiring a complete code rewrite. We adopt a task-based paradigm, similarly to OmpSS [3] and the latest version of OpenMP [9]. Task-based models offer a straightforward way to express communication across tasks, by explicitly defining inter-task data dependencies. Parallelism is expressed by the programmer in the form of independent tasks, however the scheduling of the tasks is not explicitly controlled by the programmer, but is performed at runtime, also taking into account the data dependencies among tasks.

Listing 1 illustrates the use of our programming model, using the Sobel filter as a running example.

```

1 #pragma omp task [significant(expr(...))]
2   [approxfun(function())]
3   [label(...)] [in(...)] [out(...)]

```

Listing 2: *#pragma omp task*

Tasks are specified using the *#pragma omp task* directive (Listing 2), followed by a function which is equivalent to the task body.

The significance of the task is specified through the *significant()* clause. Significance takes values in the range [0.0, 1.0] and characterizes the relative importance of tasks for the quality of the end-result of the application. Depending on their (relative) significance, tasks may be approximated or dropped at runtime. The special values 1.0 and 0.0 are used for tasks that must *unconditionally* be executed accurately and approximately, respectively.

For tasks with significance less than 1.0, the programmer may provide an alternative, approximate task body, through the *approxfun()* clause. This function is executed whenever the runtime opts for a non-accurate computation of the task. It typically implements a simpler, approximate version of the computation, which may even degenerate to just setting default values to the output. If a task is selected by the runtime system to be executed approximately, and the programmer has not supplied an *approxfun* version, it is simply dropped by the runtime. It should be noted that the *approxfun* function implicitly takes the same arguments as the function implementing the accurate version of the task body.

Programmers explicitly specify data flow to the task through the *in()* and *out()* clauses. This information is exploited by the runtime to automatically determine the dependencies among tasks.

Finally, *label()* can be used to group tasks, and to assign the group a common identifier (name), which is in turn used as a reference to implement synchronization at the granularity of task groups (see next).

For example, in lines 51- 56 of Listing 1 a separate task is created to compute each row of the output image. The significance of the tasks ranges between 0.1 and 0.9 in a round-robin way (line 53),

```

1 int sblX(const unsigned char img[], int y, int x) {
2     return img[(y-1)*WIDTH+x-1]
3         + 2*img[y*WIDTH+x-1] + img[(y+1)*WIDTH+x-1]
4         - img[(y-1)*WIDTH+x+1]
5         - 2*img[y*WIDTH+x+1] - img[(y+1)*WIDTH+x+1];
6 }
7
8
9 int sblX_appr(const unsigned char img[],
10              int y, int x) {
11     return /* img[(y-1)*WIDTH+x-1] Ommited taps */
12         + 2*img[y*WIDTH+x-1] + img[(y+1)*WIDTH+x-1]
13         /* - img[(y-1)*WIDTH+x+1] Ommited taps */
14         - 2*img[y*WIDTH+x+1] - img[(y+1)*WIDTH+x+1];
15 }
16
17 /* sblY and sblY_appr are similar */
18
19 void sbl_task(unsigned char res[],
20              const unsigned char img[], int i) {
21     unsigned int p, j;
22
23     for (j=1; j<WIDTH-1; j++) {
24         p = sqrt(pow(sblX(img, i, j),2) +
25                 pow(sblY(img, i, j),2));
26         res[i*WIDTH + j] = (p > 255) ? 255 : p;
27     }
28 }
29
30
31 void sbl_task_appr(unsigned char res[],
32                  const unsigned char img[], int i) {
33     unsigned int p, j;
34
35     for (j=1; j<WIDTH-1; j++) {
36         /* abs instead of pow/sqrt,
37            approximate versions of sblX, sblY */
38         p = abs(sblX_appr(img, i, j) +
39                sblY_appr(img, i, j));
40         res[i*WIDTH + j] = (p > 255) ? 255 : p;
41     }
42 }
43
44
45 double sobel(void) {
46     int i;
47     unsigned char img[WIDTH*HEIGHT], res[WIDTH*HEIGHT];
48
49     /* Initialize img array and reset res array */
50     ...
51     for (i=1; i<HEIGHT-1; i++)
52         #pragma omp task label(sobel) in(img) out(res) \
53         significant((i%9 + 1)/10.0) approxfun(sbl_task_appr)
54         sbl_task(res, img, i); /* Compute a single
55                                output image row */
56     }
57     #pragma omp taskwait label(sobel) ratio(0.35)
58 }

```

Listing 1: Programming model use case: Sobel filter

which ensures that there will not be extreme, apprehensible quality fluctuations across different areas of the output image. Care has also been taken in this case to avoid using the special values 0.0 and 1.0. Moreover, an approximate version of the task body is implemented by the *sbl_task_appr* function (lines 31–42). This function implements a light-weight version of the computation, substituting complex arithmetic operations with simpler ones (line 38), while at the same time skipping some filter taps (lines 11, 13). All tasks created in the specific loop belong to the *sobel* task group, using *img* as input and *res* as output (line 52).

```

1 #pragma omp taskwait [on(...)] [label(...)]
2 [ratio(...)]

```

Listing 3: #pragma omp taskwait

The proposed programming model supports explicit barrier-type synchronization through the *#pragma omp taskwait* directive (Listing 3). A *taskwait* can serve as a global barrier, instructing the runtime to wait for all tasks spawned up to that point in the code. Alternatively, it can implement a barrier at the granularity of a specific task group, if the *label()* clause is present; in this case the runtime system waits for the termination of all tasks of that group. Finally, the *on()* clause can be used to instruct the runtime to wait for all tasks that affect a specific variable or data construct.

Furthermore, the *omp taskwait* barrier can be used to control the minimum quality of application results. Through the *ratio()* clause, the programmer can instruct the runtime to execute (at least) the specified percentage of all tasks – either globally or in a specific group, depending on the existence of the *label()* clause – in their accurate version, while *respecting* task significance (i.e., a more significant task should not be executed approximately, while a less significant task is executed accurately). The ratio takes values in the range [0.0, 1.0] and serves as a single, straightforward knob to enforce a minimum quality in the performance / quality / energy optimization space. Smaller ratios give the runtime more energy reduction opportunities, however at a potential quality penalty.

For example, line 57 of Listing 1 specifies a barrier for the tasks of the *sobel* task group. The runtime needs to ensure that at least the 35% most significant tasks of the group will be executed accurately.

The compiler for the programming model is implemented based on a source-to-source compiler infrastructure [26]. It recognizes the pragmas introduced by the programmer and lowers them to corresponding calls of the runtime system discussed in Section 3.

3. Runtime

We demonstrate how to extend existing runtime systems to support our programming model for approximate computing. To this end, we extend a task-based parallel runtime system that implements OpenMP 4.0-style task dependencies [23].

Our runtime system is organized as a master/slave work-sharing scheduler. The master thread starts executing the main program sequentially. For every task call encountered, the task is enqueued in a per-worker task queue. Tasks are distributed across workers in round-robin fashion. Workers select the oldest tasks from their queues for execution. When a worker’s queue runs empty, the worker may steal tasks from other worker’s queues.

The runtime system furthermore implements an efficient mechanism for identifying and enforcing dependencies between tasks that arise from annotations of the side effects of tasks with *in(...)* and *out(...)* clauses. Dependence tracking is however not affected by our approximate computing programming model. As such, we provide no further details on this feature.

3.1 Runtime API Extension

The runtime exposes an API that matches with the pragma-based programming model. Every pragma in the program is translated in one or more runtime calls. The runtime API is extended to convey the new information in the programming model. Task creation is extended to indicate the *task group* and *significance* of the task, as well as an alternative (approximate) task function. On the first use of a task group, the compiler inserts a call to *tpc_init_group()* to create support data structures in the runtime for the task group. This API call also conveys the per-group ratio of tasks that must be executed accurately.

```

1 TaskDesc buffer[BUFFER_SIZE]; // to analyze tasks
2 size_t task_count = 0;        // buffer occupation
3 float group_ratio;            // set by #pragma
4
5 void buffer_task(TaskDesc t) { // called by master
6     thread
7     buffer[task_count] = t;
8     task_count++;
9     if (task_count == BUFFER_SIZE)
10        flush_buffer();
11 }
12
13 void flush_buffer() { // when tasks need to execute
14     sort(buffer); // sort by increasing significance
15     for (i=0; i<task_count; i++) {
16         if (i < group_ratio * task_count)
17             issue_accurate_task(buffer[i]);
18         else
19             issue_approximate_task(buffer[i]);
20     }
21     task_count = 0;
22 }

```

Listing 4: Global task buffering policy to choose the accuracy of a task

An additional waiting API call is created. Next to the API call *tpc_wait_all()*, which waits for all tasks to finish, we create the API call *tpc_wait_group()* to synchronize on the completion of a task group.

3.2 Runtime Support for Approximate Computing

The job of the runtime system is to selectively execute a subset of the tasks approximately while respecting the constraints given by the programmer. The relevant information consists of (i) the significance of each task, (ii) the group a task belongs to, and (iii) the fraction of tasks that may be executed approximately for each task group. Moreover, preference should be given to approximating tasks with lower significance values as opposed to tasks with high significance values.

The runtime system has no a priori information on how many tasks will be issued in a task group, nor what the distribution is of the significance levels in each task group. This information must be collected at runtime. In the ideal case, the runtime system knows this information in advance. Then, it is straightforward to execute approximately those tasks with the lowest significance in each task group. The policies we design must however work without this information, and estimate it at runtime. We define two policies, one globally controlled policy based on buffering issued tasks and analyzing their properties, and a policy that estimates the distribution of significance levels using per-worker local information.

3.3 Global Task Buffering (GTB)

In the first policy the master thread buffers a number of tasks as it creates them, postponing the issue of the tasks in the worker queues. When the buffer is full, or when the call to *tpc_wait_all()* or *tpc_wait_group()* is made, the tasks in the buffer are analyzed and sorted by significance. Given a per-group ratio of accurate tasks R_g , and a number of B tasks in the buffer, then the $R_g \cdot B$ tasks with the highest significance level are executed accurately. The tasks are subsequently issued to the worker queues. The policy is described in Listing 4 for a single task group. The variables described (buffer, task count and per-group accuracy ratio) are replicated over all task groups introduced by the programmer.

The task buffering policy is parameterized by the task buffer size. A larger buffer size allows the runtime to take more informed decisions. Notably, if the buffer size is sufficiently large, the run-

time can end up buffering all tasks until the corresponding synchronization barrier is encountered, and thus take a fully correct decision as to which tasks to run accurately/approximately. In our implementation, the buffer size is a configurable parameter passed to the runtime system at compile time.

The global buffer policy has the potential disadvantage that it slows down the program by postponing task execution until the buffer is full and sorted. In the extreme case, the runtime system needs to wait for all tasks to be issued and sorted in the buffer before starting their execution. This overhead can be mitigated by using a smaller window size and tasks of coarse enough granularity, so that the runtime system can overlap task issue with task execution. Using smaller window sizes will incur the cost of not making fully correct decisions for approximate execution. Section 4 demonstrates that the GTB policy sustains low overhead in practice.

3.4 Local Queue History (LQH)

The local queue history policy avoids the step of task buffering. Tasks are issued to worker queues immediately as they are created. The worker decides whether to approximate a task right before it starts its execution, based on the distribution of significance levels of the tasks executed so far, and the target ratio of accurate tasks (supplied by the programmer). Hereto, the workers track the number of tasks at each significance level as they are executed.

Formally, the local queue history policy operates as follows. Let $t_g(s)$ indicate the number of tasks in task group g observed by a worker with significance s or less. These statistics are updated for every executed task. Note that the significance levels s are constrained to the range 0.0 to 1.0. In the runtime system, we implement 101 discrete (integer) levels to simplify the implementation, ranging from 0.0 to 1.0 (inclusive) in steps of 0.01. By construction, $t_g(1.0)$ equals the total number of tasks executed so far. Let R_g be the target ratio of tasks that should be executed accurately in task group g , as set by the programmer. Then, assuming a task has significance level s , it is executed accurately if $t_g(s) > (1 - R_g)t_g(1.0)$, otherwise it is executed approximately.

This policy attempts to achieve a ratio of accurately executed tasks that converges to R_g and also approximates those tasks with the lowest significance level, as stipulated by the programming model.

The local queue history algorithm is performed independently by each worker using only local information from the tasks that appear in their work queue. Tasks of one group are distributed among the workers via pushing of tasks to different local queues by the master and work-stealing. As a result, each worker has only partial information about each group.

The overhead of the local queue history algorithm is the book-keeping of the statistics that form the execution history of a group. This happens every time a task is executed. Updating statistics includes accessing an array of size equal to the number of distinct significance levels (101 in the runtime), which is negligible compared to the granularity of the task.

The local queue history algorithm requires no global snapshot of all tasks in the program and no synchronization between workers and the master. It is thus more realistic and scalable than global task buffering. However, given that each worker has only a localized view of the tasks issued, the runtime system can only approximately enforce the quality requirements set by the programmer.

4. Experimental Evaluation

We performed a set of experiments to investigate the performance of the proposed programming model and runtime policies, using different benchmark codes that were re-written using the task-based pragma directives. In particular, we evaluate our approach in terms

Benchmark	Approximate or Drop	Approx Degree			Quality
		Mild	Med	Aggr	
Sobel	A	80%	30%	0%	PSNR
DCT	D	80%	40%	10%	PSNR
MC	D, A	100%	80%	50%	Rel. Err.
Kmeans	A	80%	60%	40%	Rel. Err.
Jacobi	D, A	10^{-4}	10^{-3}	10^{-2}	Rel. Err.
Fluidanimate	A	50%	25%	12.5%	Rel. Err.

Table 1: Benchmarks used for the evaluation. For all cases, except Jacobi, the approximation degree is given by the percentage of accurately executed tasks. In Jacobi, it is given by the error tolerance in convergence of the accurately executed iterations/tasks (10^{-5} in the native version).

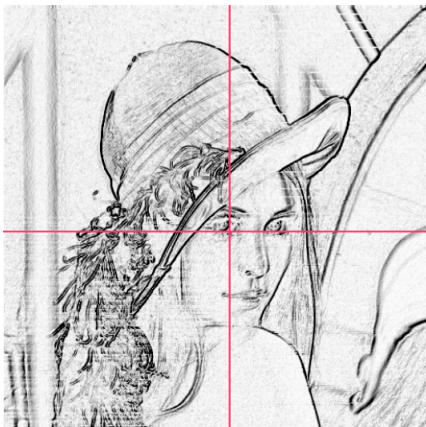


Figure 1: Different levels of approximation for the Sobel benchmark

of: (i) The potential for performance and energy reduction; (ii) The potential to allow graceful quality degradation; (iii) The overhead incurred by the runtime mechanisms. In the sequel, we introduce the benchmarks and the overall evaluation approach, and discuss the results achieved for various degrees of approximation under different runtime policies.

4.1 Approach

We use a set of six benchmarks, outlined in Table 1, where we apply different approximation approaches, subject to the nature/characteristics of the respective computation.

Sobel is a 2D filter used for edge detection in images. The approximate version of the tasks uses a lightweight Sobel stencil with just 2/3 of the filter taps. Additionally, it substitutes the costly formula $\sqrt{sbl_x^2 + sbl_y^2}$ with its approximate counterpart $|sbl_x| + |sbl_y|$. The way of assigning significance to tasks ensures that the approximated pixels are uniformly spread throughout the output image.

Discrete Cosine Transform (*DCT*) is a module of the JPEG compression and decompression [20] algorithm. We assign higher significance to tasks that compute lower frequency coefficients.

MC [24] applies a Monte Carlo approach to estimate the boundary of a subdomain within a larger partial differential equation (PDE) domain, by performing random walks from points of the subdomain boundary to the boundary of the initial domain. Approximate configurations drop a percentage of the random walks and the corresponding computations. A modified, more lightweight

methodology is used to decide how far from the current location the next step of a random walk should be.

K-means clustering aims to partition n observations in a multi-dimensional space into k clusters by minimizing the distance of cluster members to a cluster representative. In each iteration the algorithm spawns a number of tasks, each being responsible for a subset of the entire problem. All tasks are assigned the same significance value. The degree of approximation is controlled by the *ratio* used at *taskwait* pragmas. Approximated tasks compute a simpler version of the euclidean distance, while at the same time considering only a subset (1/8) of the dimensions. Only accurate results are considered when evaluating the convergence criteria.

Jacobi is an iterative solver of diagonally dominant systems of linear equations. We execute the first 5 iterations approximately, by dropping the tasks (and computations) corresponding to the upper right and lower left areas of the matrix. This is not catastrophic, due to the fact that the matrix is diagonally dominant and thus most of the information is within a band near the diagonal. All the following steps, until convergence, are executed accurately, however at a higher target error tolerance than the native execution (see Table 1).

Fluidanimate, a code from the PARSEC benchmark suite [2], applies the smoothed particle hydrodynamics (SPH) method to compute the movement of a fluid in consecutive time steps. The fluid is represented as a number of particles embedded in a grid. Each time step is executed as either fully accurate or fully approximate, by setting the *ratio* clause of the *omp taskwait* pragma to either 0.0 or 1.0. In the approximate execution, the new position of each particle is estimated assuming it will move linearly, in the same direction and with the same velocity as it did in the previous time steps.

Three different degrees of approximation are studied for each benchmark: *Mild*, *Medium*, and *Aggressive* (see Table 1). They correspond to different choices in the quality vs. energy and performance space. No approximate execution led to abnormal program termination. It should be noted that, with the partial exception of Jacobi, quality control is possible solely by changing the *ratio* parameter of the *taskwait* pragma. This is indicative of the flexibility of our programming model. As an example, Figure 1 visualizes the results of different degrees of approximation for *Sobel*: the upper left quadrant is computed with no approximation, the upper right is computed with *Mild* approximation, the lower left with *Medium* approximation, whereas the lower right corner is produced when using *Aggressive* approximation.

The quality of the final result is evaluated by comparing it to the output produced by a fully accurate execution of the respective code. The appropriate metric for the quality of the final result differs according to the computation. For benchmarks involving image processing (*DCT*, *Sobel*), we use the peak signal to noise ratio (*PSNR*) metric, whereas for *MC*, *Kmeans*, *Jacobi* and *Fluidanimate* we use the relative error.

In the experiments, we measure the performance of our approach for the different benchmarks and approximation degrees, for the two different runtime policies GTB and LQH. For GTB, we investigate two cases: the buffer size is set so that tasks are buffered until the synchronization barrier (referred to as Max Buffer GTB); the buffer size is set to a smaller value, depending on the computation, so that task execution can start earlier (referred to as GTB).

As a reference, we compare our approach against:

- A fully accurate execution of each application, using a significance agnostic version of the runtime system.
- An execution using loop perforation [19], a simple yet usually effective compiler technique for approximation. Loop perforation is also applied in three different degrees of aggressiveness.

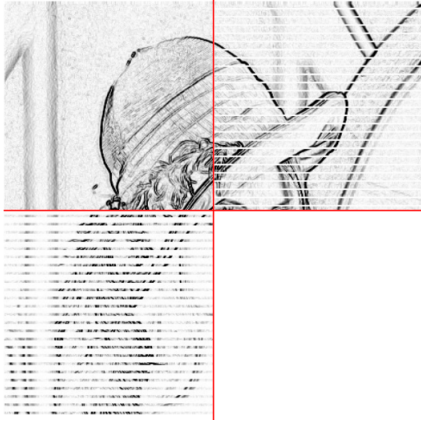


Figure 3: Different levels of perforation for the Sobel benchmark. Accurate execution, Perforation of 20%, 70% and 100% of loop iterations on the upper left, upper right, lower left and lower right quadrants respectively.

The perforated version executes the same number of tasks as those executed accurately by our approach.

The experimental evaluation is carried out on a system equipped with 2 *Intel(R) Xeon(R) CPU E5-2650* processors clocked at 2.00 GHz, with 64 GB shared memory. Each CPU consists of 8 cores. Although cores support SMT execution (hyper-threading), we deactivated this feature during our experiments. We use Centos 6.5 Linux Operating system with the 2.6.32 Linux kernel. Each execution pinned 16 threads on all 16 cores.

Finally the energy and power are measured using likwid [22] to access the Running Average Power Limit (RAPL) registers of the processors.

4.2 Experimental Results

Figure 2 depicts the results of the experimental evaluation of our system. For each benchmark we present execution time, energy consumption and the corresponding error metric.

The approximated versions of the benchmarks execute significantly faster and with less energy consumption compared to their accurate counterparts. Although the quality of the application output deteriorates as the approximation level increases, this is typically done in a graceful manner, as it can be observed in Figure 1 and the ‘Quality’ column of Figure 2.

The GTB policies with different buffer sizes are comparable with each other. Even though Max buffer GTB postpones task issue until the creation of all tasks in the group, this does not seem to penalize the policy. In most applications tasks are coarse-grained and are organized in relatively small groups, thus minimizing the task creation overhead and the latency for the creation of all tasks within a group. LQH is typically faster and more energy-efficient than both GTB flavors, except for *Kmeans*.

In the case of *Sobel*, the perforated version seems to significantly outperform our approach in terms of both energy consumption and execution time. However the cost of doing so is unacceptable output quality, even for the mild approximation level as shown in Figure 3. Our programming model and runtime policies achieve graceful quality degradation, resulting in acceptable output even with aggressive approximation, as illustrated in Figure 1.

DCT is friendly to approximations: it produces visually acceptable results even if a large percentage of the computations is dropped. Our policies, with the exception of the Max Buffer version of GTB, perform comparably to loop perforation in terms of per-

formance and energy consumption, yet resulting in higher quality results¹. This is due to the fact that our model offers more flexibility than perforation in defining the relative significance of code regions in DCT. The problematic performance of GTB(Max Buffer) is discussed later in this Section, when evaluating the overhead of the runtime policies and mechanisms.

The approximate version of *MC* significantly outperforms the original accurate version, without suffering much of a penalty on its output quality. Randomized algorithms are inherently susceptible to approximations without requiring much sophistication. It is characteristic that the performance of our approach is almost identical to that of blind loop perforation. We observe that the LQH policy attains slightly better results. In this case, we found that the LQH policy undershoots the requested ratio, evidently executing fewer tasks². This affects quality, which is lower than that achieved by the rest of the policies.

Kmeans behaves gracefully as the level of approximation increases. Even in the aggressive case, all policies demonstrate relative errors less than 0.45%. The GTB policies are superior in terms of execution time and energy consumption in comparison with the perforated version of the benchmark. Noticeably, the LQH policy exhibits slow convergence to the termination criteria. The application terminates when the number of objects which move to another cluster is less than 1/1000 of the total object population. As mentioned in the Section 4.1, objects which are computed approximately do not participate in the termination criteria. GTB policies behave deterministically, therefore always selecting tasks corresponding to specific objects for accurate executions. On the other hand, due to the effects dynamic load balancing in the runtime and its localized perspective, LQH tends to evaluate accurately different objects in each iteration. Therefore, it is more challenging for LQH to achieve the termination criterion. Nevertheless, LQH produces results with the same quality as a fully accurate execution with significant performance and energy benefits.

Jacobi is a particular application, in the sense that approximations can affect its rate of convergence in deterministic, yet hard to predict and analyze ways. The blind perforation version requires fewer iterations to converge, thus resulting in lower energy consumption than our policies. Interestingly enough, it also results in a solution closer to the real one, compared with the accurate execution.

The perforation mechanism could not be applied on top of the *Fluidanimate* benchmark. This is because if the evaluation of the movement of part of the particles during a time-step is totally dropped, the physics of the fluid are violated leading to completely wrong results. Our programming model offers the programmer the expressiveness to approximate the movement of the liquid for a set of time-steps. Moreover, in order to ensure stability, it is necessary to alternate accurate and approximate time steps. In our programming model this is achieved in a trivial manner, by alternating the parameter of the *ratio* clause at *taskbarrier* pragmas between 100% and the desired value in consecutive time steps. It is worth noting that *Fluidanimate* is so sensitive to errors that only the mild degree of approximation leads to acceptable results. Even so, the LQH policy requires less than half the energy of the accurate execution, with the 2 versions of the GTB policy being almost as efficient.

Following, we evaluate the overhead of the runtime policies and mechanisms discussed in Sections 3.3 and 3.4. We measure the performance of each benchmark when executed with a significance-agnostic version of the runtime system, which does not include the execution paths for classifying and executing tasks according

¹ Note that PSNR is a logarithmic metric

² 4.6% and 5.1% more that requested tasks are approximated for the aggressive and the medium case respectively.

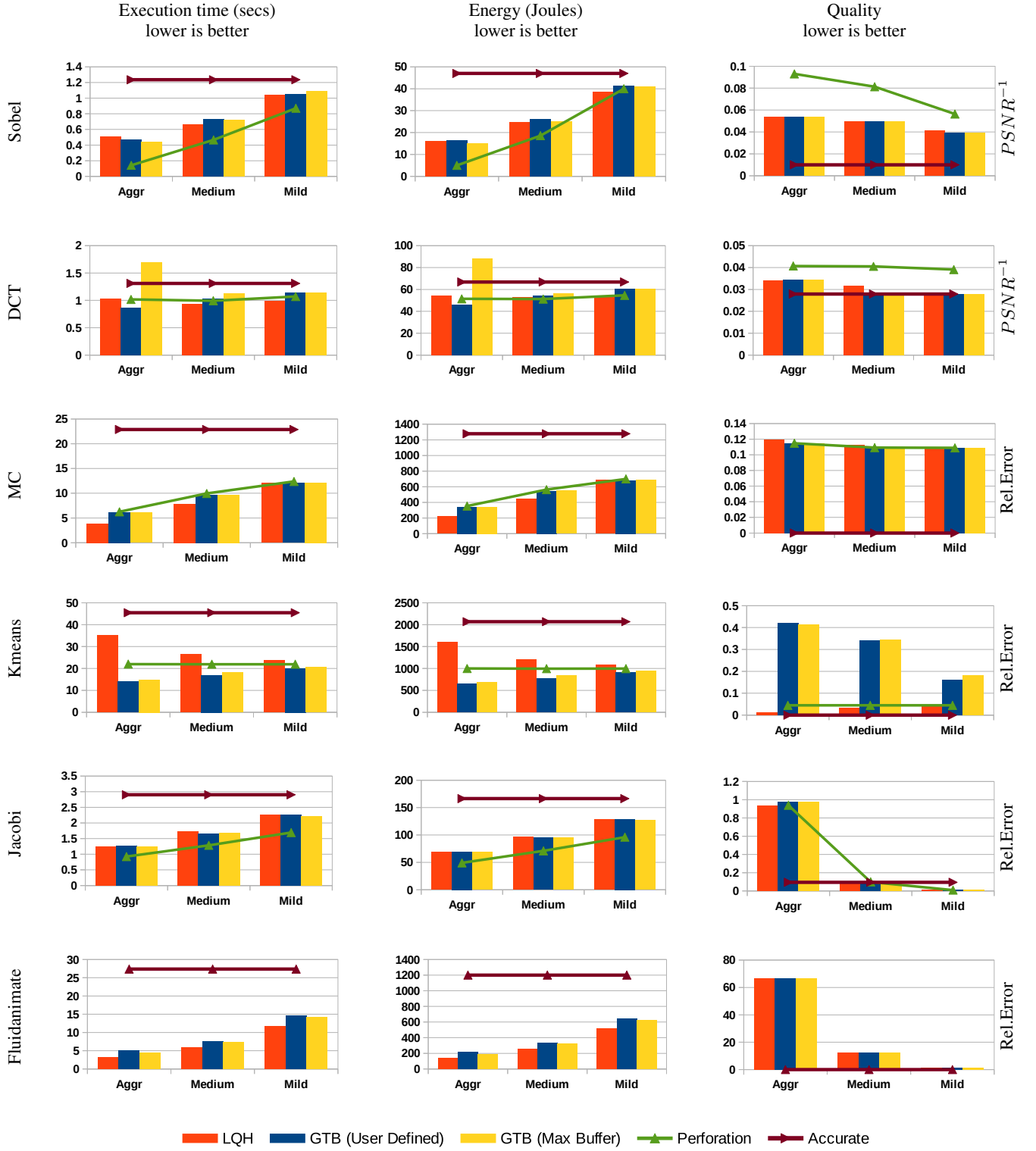


Figure 2: Execution time, energy and quality of results for the benchmarks used in the experimental evaluation under different runtime policies and degrees of approximation. In all cases lower is better. Quality is depicted as $PSNR^{-1}$ for Sobel and DCT, relative error (%) is used in all others benchmarks. The accurate execution and the approximate execution using perforation are visualized as lines. Note that perforation was not applicable for Fluidanimate.

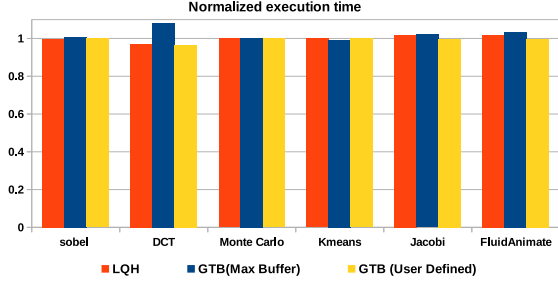


Figure 4: The normalized execution time of benchmarks under different task categorization policies, with respect to that over the significance-agnostic runtime system

Benchmark	Inversed Significance Tasks (%)			Average Ratio Diff		
	LQH	GTB(UD)	GTB (MB)	LQH	GTB(UD)	GTB (MB)
Sobel	2.7	0	0	0.07	0	0
DCT	2.7	0	0	0.18	0	0
MC	4.8	0	0	0.17	0	0
KMeans	0	0	0	0.9	0	0
Jacobi	0	0	0	0.12	0	0
FluidAnimate	0	0	0	0	0	0

Table 2: Degree of accuracy of the proposed policies.

to significance. We then compare it with the performance attained when executing the benchmarks with the significance-aware version of the runtime. All tasks are created with the same significance and the ratio of tasks executed accurately is set to 100%, therefore eliminating any benefits of approximate execution. Figure 4 summarizes the results. It is evident that the significance-aware runtime system typically incurs negligible overhead. The overhead reaches in the order of 7% in the worst case (DCT under the GTB Max Buffer policy). DCT creates many lightweight tasks, therefore stressing the runtime. At the same time, given that for DCT task creation is a non-negligible percentage of the total execution time, the latency between task creation and task issue introduced by the Max Buffer version of the GTB policy results in a measurable overhead.

The last step of our evaluation focuses on the accuracy of the policies in terms of respecting the significance of tasks and the user-supplied ratio of accurate tasks to be executed. Table 2 summarizes the results. The average offset in the ratio of accurate tasks executed is calculated by the following formula:

$$ratio_diff = \frac{\sum_{i=1}^{Groups} |requestedratio_i - providedratio_i|}{TotalGroups}$$

The two versions of GTB respect perfectly task significance and the user-specified ratio. This is totally expected for the Max Window version of GTB. The version of GTB using a limited window benefits by the relatively small task groups created by the applications and the smoothly distributed significance values in tasks of each group. LQH, in turn, is inherently more inaccurate, due to its localized perspective. It manages to avoid significance inversion only in cases where all tasks within each task group have the same significance (*Kmeans*, *Jacobi*, *Fluidanimate*). Even in these cases, LQH may slightly deviate from the specified ratio, due to the loose collaboration of policy modules active on different workers.

5. Related Work

We classify related work in approximate computation into general-purpose frameworks, parallel programming and execution models

that implement approximate computation, and other approaches, including domain-specific frameworks and hardware support for approximate computation. Finally, we review prior work on runtime energy optimization of parallel programs, that does not employ approximation.

5.1 General-Purpose Approximation Frameworks

Several frameworks for approximate computing discard parts of code at runtime, while asserting that the quality of the result complies with quality criteria provided by the programmer. Green [1] is an API for loop-level and function approximation. Loops are approximated with a reduction of the loop trip count. Functions are approximated with multi-versioning. The API includes calibration functions that build application-specific QoS models for the outputs of the approximated blocks of code, as well as re-calibration functions for correcting unacceptable errors that may incur due to approximation. Sloan et al. [21] provide guidelines for manual control of approximate computation and error checking in software. These frameworks delegate the control of approximate code execution to the programmer. We explore an alternative approach where the programmer uses a higher level of abstraction for approximation, namely computational significance, while the system software translates this abstraction into energy- and performance-efficient approximate execution.

Loop perforation [19] is a compiler technique that classifies loop iterations into critical and non-critical ones. The latter can be dropped, as long as the results of the loop are acceptable from a quality standpoint. Input sampling and code versioning [28] also use the compiler to selectively discard inputs to functions and substitute accurate function implementations with approximate ones. Similarly to loop perforation and code versioning, our framework benefits from task dropping and the execution of approximate versions of tasks. However, we follow a different approach whereby these optimizations are driven from user input on the relative significance of code blocks and are used selectively in the runtime system to meet user-defined quality criteria.

EnerJ [16] implements approximate data types and supports user-defined “approximable” methods, without tying these abstractions to a specific approximate execution model. To achieve energy savings, the prototype implementation of EnerJ uses a simulated environment where it stores approximate data types in DRAM with low refresh rate and SRAM with low supply voltage. Approximable methods are executed on aggressively voltage-scaled processors, with ISA extensions for approximation [4, 17]. Similarly to our framework, EnerJ provides abstractions that allow the programmer to provide hints on where approximate execution can be safely used in a program. Contrary to our framework, EnerJ does not use a runtime substrate for approximation on general-purpose hardware and does not consider code dropping or task-parallel execution.

5.2 Parallel Approximation Frameworks

Quickstep [8] is a tool that approximately parallelizes sequential programs. The parallelized programs are subjected to statistical accuracy tests for correctness. Quickstep tolerates races that occur after removing synchronization operations that would otherwise be necessary to preserve the semantics of the sequential program. Quickstep thus exposes additional parallelization and optimization opportunities via approximating the data and control dependencies in a program. On the other hand, QuickStep does not enable algorithmic and application-specific approximation, which is the focus of our work.

Variability-aware OpenMP [11] is a set of OpenMP extensions that enable a programmer to specify blocks of code that can be computed approximately. The programmer may also specify error tolerance in terms of the number of most significant bits in a vari-

able which are guaranteed to be correct. Variability-aware OpenMP applies approximation only to specific FPU operations, which execute on specialized FPUs with configurable accuracy. Our framework applies selective approximation at the granularity of tasks, using the significance abstraction. Our programming and execution model thus provides additional flexibility to drop or approximate code, while preserving output quality. Furthermore, our framework does not require specialized hardware support.

Variation-tolerant OpenMP [10] uses a runtime system that characterizes OpenMP tasks in terms of their vulnerability to errors. The runtime system assesses error vulnerability of tasks online, similarly to our LQH policy for significance characterization. The variation-tolerant OpenMP runtime uses a hardware error counter to apportion errors to tasks and estimate task vulnerability to errors. The scheduler is a variant of an FCFS, centralized scheduler that uses task vulnerability to select the cores on which each task runs, in order to minimize the number of instructions that are likely to incur errors. Variation-tolerant OpenMP does not consider explicitly identified approximate code and its selective execution for quality-aware energy and performance optimization.

5.3 Other Approximation Frameworks

Several software and hardware schemes for approximate computing follow a domain-specific approach. ApproxIt [27] is a framework for approximate iterative methods, based on a lightweight quality control mechanism. Unlike our task-based approach, ApproxIt uses coarse-grain approximation at a minimum granularity of one solver iteration. Gschwandtner et al. use a similar iterative approach to execute error-tolerant solvers on processors that operate with near-threshold voltage (NTC) and reduce energy consumption by replacing cores operating at nominal voltage with NTC cores [6]. Schmoll et al. [18] present algorithmic and static analysis techniques to detect variables that must be computed reliably and variables that can be computed approximately in an H.264 video decoder. Although we follow a domain-agnostic approach in our approximate computing framework, we provide sufficient abstractions for implementing the aforementioned application-specific approximation methods.

SAGE [14] is a compiler and runtime environment for automatic generation of approximate kernels in machine learning and image processing applications. Paraprox [15] implements transparent approximation for data-parallel programs by recognizing common algorithmic kernels and replacing them with approximate equivalents. ASAC [12] provides sensitivity analysis for automatically generated code annotations that quantify significance. We do not explore automatic generation of approximate code in this work. However, our techniques for quality-aware, selective execution of approximate code are directly applicable to scenarios where the approximate code is derived from a compiler, instead of source code annotations.

Hardware support for approximate computation has taken the form of programmable vector processors [25], neural networks that approximate the results of code regions in hardware [5], and low-voltage probabilistic storage [13]. These frameworks assume non-trivial, architecture-specific support from the system software stack, whereas we depend only on compiler and runtime support for task-parallel execution, which is already widely available on commodity multi-core systems. ESRA [7] is a multi-core architecture where cores are either fully reliable or have relaxed reliability. Programs running on ESRA divide their code into critical (typically control code) and non-critical (typically data processing code) parts and assign these to reliable or unreliable cores, respectively. Therefore, ESRA uses an explicit and application-specific assignment of code to cores with different levels of reliability. We follow a different approach whereby the programmer uses signifi-

cance to implicitly indicate code that can be approximated and the runtime system implements selective approximation. In our framework, accurate and approximate code may run on any core for load balancing purposes.

6. Conclusions

We introduced a programming model that supports approximate computing at the granularity of tasks. Tasks are widely used to express parallelism in a high-level and platform-neutral way. We believe that tasks can also be used to introduce approximate versions of specific parts of the computation in a structured way that is amenable to flexible runtime scheduling to achieve energy-efficient program execution at a controllable degradation of output quality.

We also introduced extensions to a task-based runtime system to exploit significance information, along with a set of significance-centric scheduling policies for elastically deciding which tasks to execute accurately and which approximately, while at the same time respecting programmer's specifications.

We have performed a first evaluation of our implementation on an Intel-based multiprocessor consisting of twin multicore sockets. The results across several different benchmark codes are encouraging, and show that the programmer can easily target different energy-quality trade-offs, by adjusting in the majority of cases a single parameter: the percentage of tasks to execute accurately.

In the future, we wish to explore more optimization scenarios, such as DFVS in conjunction with suitable runtime policies for executing approximate (and more light-weight) task versions on the slower but also less power-hungry CPUs, as well as for using more such cores to make up for this slower execution. We are also interested in extending our programming model to support approximate computing on top of ultra low-power but unreliable hardware.

References

- [1] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10*, pages 198–209, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. . URL <http://doi.acm.org/10.1145/1806596.1806620>.
- [2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, PACT '08*, pages 72–81, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-282-5. . URL <http://doi.acm.org/10.1145/1454115.1454128>.
- [3] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Omppss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02): 173–193, 2011.
- [4] H. Esmailzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII*, pages 301–312, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. . URL <http://doi.acm.org/10.1145/2150976.2151008>.
- [5] H. Esmailzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-45*, pages 449–460, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4924-8. . URL <http://dx.doi.org/10.1109/MICRO.2012.48>.
- [6] P. Gschwandtner, C. Chaliou, D. Nikolopoulos, H. Vandierendonck, and T. Fahringer. On the potential of significance-driven execution for energy-aware hpc. *Computer Science - Research and Development*,

- pages 1–10, 2014. ISSN 1865-2034. . URL <http://dx.doi.org/10.1007/s00450-014-0265-9>.
- [7] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra. Ersar: Error resilient system architecture for probabilistic applications. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1560–1565, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association. ISBN 978-3-9810801-6-2. URL <http://dl.acm.org/citation.cfm?id=1870926.1871302>.
 - [8] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM Trans. Embed. Comput. Syst.*, 12(2s):88:1–88:26, May 2013. ISSN 1539-9087. . URL <http://doi.acm.org/10.1145/2465787.2465790>.
 - [9] OpenMP Architecture Review Board. OpenMP Application Program Interface (version 4.0). Technical report, July 2013.
 - [10] A. Rahimi, A. Marongiu, P. Burgio, R. K. Gupta, and L. Benini. Variation-tolerant openmp tasking on tightly-coupled processor clusters. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 541–546, San Jose, CA, USA, 2013. EDA Consortium. ISBN 978-1-4503-2153-2. URL <http://dl.acm.org/citation.cfm?id=2485288.2485422>.
 - [11] A. Rahimi, A. Marongiu, R. K. Gupta, and L. Benini. A variability-aware openmp environment for efficient execution of accuracy-configurable computation on shared-fpu processor clusters. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '13, pages 35:1–35:10, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4799-1417-3. URL <http://dl.acm.org/citation.cfm?id=2555692.2555727>.
 - [12] P. Roy, R. Ray, C. Wang, and W. F. Wong. Asac: Automatic sensitivity analysis for approximate computing. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, pages 95–104, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2877-7. . URL <http://doi.acm.org/10.1145/2597809.2597812>.
 - [13] M. Salajegheh, Y. Wang, A. A. Jiang, E. Learned-Miller, and K. Fu. Half-wits: Software techniques for low-voltage probabilistic storage on microcontrollers with nor flash memory. *ACM Trans. Embed. Comput. Syst.*, 12(2s):91:1–91:25, May 2013. ISSN 1539-9087. . URL <http://doi.acm.org/10.1145/2465787.2465793>.
 - [14] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 13–24, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2638-4. . URL <http://doi.acm.org/10.1145/2540708.2540711>.
 - [15] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 35–50, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. . URL <http://doi.acm.org/10.1145/2541940.2541948>.
 - [16] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. . URL <http://doi.acm.org/10.1145/1993498.1993518>.
 - [17] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 25–36, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2638-4. . URL <http://doi.acm.org/10.1145/2540708.2540712>.
 - [18] F. Schmoll, A. Heinig, P. Marwedel, and M. Engel. Improving the fault resilience of an h.264 decoder using static analysis methods. *ACM Trans. Embed. Comput. Syst.*, 13(1s):31:1–31:27, Dec. 2013. ISSN 1539-9087. . URL <http://doi.acm.org/10.1145/2536747.2536753>.
 - [19] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ES-EC/FSE '11, pages 124–134, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0443-6. . URL <http://doi.acm.org/10.1145/2025113.2025133>.
 - [20] A. Skodras, C. Christopoulos, and T. Ebrahimi. The jpeg 2000 still image compression standard. *Signal Processing Magazine, IEEE*, 18(5):36–58, Sept. 2001. . URL <http://doi.org/10.1109/79.952804>.
 - [21] J. Sloan, J. Sartori, and R. Kumar. On software design for stochastic processors. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 918–923, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1199-1. . URL <http://doi.acm.org/10.1145/2228360.2228524>.
 - [22] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Parallel Processing Workshops (ICPPW)*, 2010 39th International Conference on, pages 207–216. IEEE, Sept. 2010. ISBN 978-0-7695-4157-0. . URL <http://dx.doi.org/10.1109/ICPPW.2010.38>.
 - [23] G. Tzenakis, A. Papatrifiantafyllou, H. Vandierendonck, P. Pratikakis, and D. Nikolopoulos. Bddt: Block-level dynamic dependence analysis for task-based parallelism. In C. Wu and A. Cohen, editors, *Advanced Parallel Processing Technologies*, volume 8299 of *Lecture Notes in Computer Science*, pages 17–31. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-45292-5. . URL http://dx.doi.org/10.1007/978-3-642-45293-2_2.
 - [24] M. Vavalis and G. Sarailidis. Hybrid-numerical-pde-solvers: Hybrid elliptic pde solvers. Sept. 2014. . URL <http://dx.doi.org/10.5281/zenodo.11691>.
 - [25] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Quality programmable vector processors for approximate computing. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 1–12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2638-4. . URL <http://doi.acm.org/10.1145/2540708.2540710>.
 - [26] F. S. Zakkak. Scoop: Language extensions and compiler optimizations for task-based programming models. Master's thesis, University of Crete, School of Sciences and Engineering, Computer Science Department, 2012.
 - [27] Q. Zhang, F. Yuan, R. Ye, and Q. Xu. Approxit: An approximate computing framework for iterative methods. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, DAC '14, pages 97:1–97:6, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2730-5. . URL <http://doi.acm.org/10.1145/2593069.2593092>.
 - [28] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 441–454, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. . URL <http://doi.acm.org/10.1145/2103656.2103710>.